Parallel SOR Iterative Algorithms and Performance Evaluation on a Linux Cluster

Chaoyang Zhang, Hong Lan, Yang Ye
University of Southern Mississippi
Hattiesburg, MS 39406

Brett D. Estrade
Naval Research Laboratory
NASA Stennis Space Center, MS 39529

***Abstract**: The successive over-relaxation (SOR) iterative method is an important solver for linear systems. In this paper, a parallel algorithm for the red-black SOR method with domain decomposition is investigated. The parallel SOR algorithm is designed by combining the traditional red-black SOR and row block domain decomposition technique, which reduces the communication cost and simplifies the parallel implementation. Two other iterative methods, Jacobi and Gauss-Seidel(G-S), are also implemented in parallel for comparison. The three parallel iterative algorithm are implemented in C and MPI (Message Passing Interface) for solving the Dirichlet problem on a Linux cluster with eight dual processor 2.6ghz 32 bit Intel Xeons, totaling 16 processors. The performances of the three algorithms are evaluated in terms of speedup and efficiency.*

Keywords: Parallel algorithm, successive over-relaxation (SOR) iteration, Linux cluster, message passing interface (MPI).

1. Introduction

The successive over-relaxation (SOR) iterative method is an important solver for linear systems. The SOR method is inherently sequential in its original form. To take advantage of the supercomputing resource with multiple processors, several parallel versions of the SOR method have been proposed. One of the widely used parallel versions is the multi-color SOR method which uses the multi-color ordering technique [1]. For complicated problems, two or more colors are required to define a multicolor ordering. Harrar II [2] and Melhem [3] studied how to quickly verify and generate a multicoloring ordering according to the given structure of a matrix or a grid. However, the multi-color SOR method is parallel only within the same color. For some problems such as two dimensional (2D) heat transfer described by Poisson equations, the Red-Black two-color SOR method is preferred. Yanheh [4] showed that the Red-Black SOR method is more efficient and smoother than the sequential SOR method. Xie proposed an efficient parallel SOR method (PSOR) using domain decomposition and interprocessor data communication techniques [5]. It is shown that PSOR is just the SOR method applied to a reordered linear system, so that the theory of SOR can also be applied to the analysis of PSOR. Other techniques such as pipeline of computation and communication and an optimal schedule of a feasible number of processors are studied and applied to define parallel versions of SOR for banded or dense matrices problems [6] and they can be implemented in parallel without changing the sequential SOR method. The parallel SOR method for particular parallel computers can also be found in [7].

Most of these early studies on parallel SOR methods focused on their mathematical properties and were designed for MIMD machines. Due to the demanding cost of supercomputers, Linux clusters have drawn more and more attention in higher performance computing and have been used for solving a variety of computational problems in science and engineering. The performance of SOR methods on distributed memory platforms, e.g., Linux clusters, may be quite different from that on shared memory supercomputers, such as SGI Onyx 10000. The actual performance of these algorithms depends on the hardware, interconnection of processors and implementation. To design an efficient parallel SOR method, task decomposition and task dependency must be investigated to achieve the maximum degree of concurrency and to minimize the communication cost in parallel computation. In this paper, a parallel version of the SOR method is designed based on the widely-used Red-Black SOR (RB-SOR) and row block domain decomposition. The parallel SOR method is implemented in MPI and C language and tested on a Linux cluster. The interprocess communication and task dependency are analyzed. The performance results such as convergence rate, speedup and efficiency are given and also compared with other methods such as Jacobi method and Gaussian-Seidel (G-S) method.

2. Serial and Parallel SOR Methods

The SOR method is a widely used iterative procedure to solve a linear system or a partial differential equation discretized by a finite difference method. Consider the Dirichlet problem

$$\begin{cases} u_{xx}+u_{yy}=f(x,y), & (x,y)\in\Omega \\ u(x,y)=g(x,y), & (x,y)\in\partial\Omega \end{cases} \tag{1}$$

with $\Omega\equiv(0,1)\times(0,1)$. Assuming a uniform partition, the intervals $I^x=I^y=[0,1]$ are divided into n_s subintervals $I_m^x=I_m^y, m=1,...,n_s$. It generates a uniform grid with spacing $h=\frac{1}{n_s}$ and nodal coordinates (x_i,y_i), where $x_i=(i-1)h$ and $y_i=(j-1)h,\ i,j=1,...,(n_s+1)$. The finite difference method (FDM) is used to discretize the Eq.(1). We consider the 5-point approximation to Eq.(1) on a unit square Ω_h and obtain the following finite different scheme at node (i,j)

$$u_{i,j-1}+u_{i,j+1}+u_{i-1,j}+u_{i+1,j}-4u_{i,j}=h^2f_{i,j} \quad \text{in } \Omega_h \tag{2}$$

where $f_{i,j}$ is the value of the function $f(x,y)$ at the node (i,j) and $u_{i,j}$ denotes the approximation of $u(x_i,y_i)$.

2.1 Serial iterative methods

The Eq. (2) can be rewritten as a matrix form $Au=f$ where A is a matrix and both u and f are vectors. The solution can be obtained using iterative methods such as Jacibi method, G-S method and SOR method. After making an initial guess of u, e.g., $u^{(0)}$, the Jacobi iterative method generates a sequence of approximations $u^{(k)}, k=1,2,3,...,$ to the solution. The approximation, $u^{(k+1)}$, at the (*k+1*) iteration is computed using the results $u^{(k)}$ obtained in the *k*th iteration:

$$u_{i,j}^{(k+1)}=\frac{1}{4}(u_{i,j-1}^{(k)}+u_{i,j+1}^{(k)}+u_{i-1,j}^{(k)}+u_{i+1,j}^{(k)})-h^2f_{i,j} \tag{3}$$

where $u_{i,j}^{(k+1)}$ at node(i,j) of $(k+1)$th iteration relies on the values of the four neighboring nodes$(i-1,j)$, $(i+1,j)$, $(i,j-1)$ and $(i,j+1)$ obtained at the previous *k*th iteration.

The parallel Jacobi iterative method is easy to implement in parallel, but its convergence rate is very low. It is seen that if we update the $u_{i,j}$ according to increasing values of subscripts *i* and *j*, the most current values $u_{i-1,j}^{(k+1)}$ and $u_{i,j-1}^{(k+1)}$ are already available when we compute the new update $u_{i,j}^{(k+1)}$. This suggests us to make use of the most recent values at node $(i-1,j)\,(i,j-1)$ to update $u_{i,j}^{(k+1)}$ and it results in G-S iterative method

$$u_{i,j}^{(k+1)}=\frac{1}{4}(u_{i,j-1}^{(k+1)}+u_{i,j+1}^{(k)}+u_{i-1,j}^{(k+1)}+u_{i+1,j}^{(k)})-h^2f_{i,j} \tag{4}$$

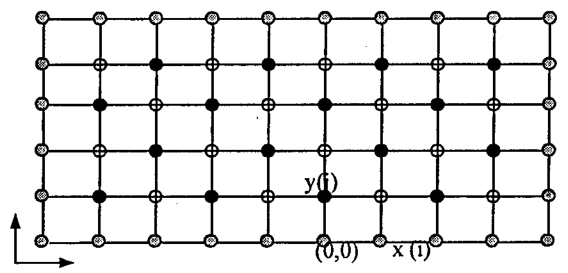
The convergence rate of G-S iterative method can be further improved by applying SOR iteration. For any $\omega\neq0$, Eq. (4) can be rewritten as

$$u_{i,j}^{(k+1)}=u_{i,j}^{(k)}+\frac{\omega}{4}(u_{i,j-1}^{(k+1)}+u_{i,j+1}^{(k)}+u_{i-1,j}^{(k+1)}+u_{i+1,j}^{(k)}-4u_{i,j}^{(k)}-h^2f_{i,j}) \tag{5}$$

in which most recently computed values $u_{i-1,j}^{(k+1)}$ and $u_{i,j-1}^{(k+1)}$ are used as soon as they are available. The optimal value of ω lies in (0, 2). The choice of $\omega = 1$ corresponds to the Gauss-Seidel iteration.

2.2 Red-Black SOR method

Eq. (5) can be implemented in parallel using the red-black ordering technique. The node (i, j) is denoted red or black according to whether $i + j$ is odd or even. If $i + j$ is odd, the node (i, j) is marked red, and if $i + j$ is even, the node (i, j) is marked black. The red-black ordering is illustrated in Fig. 1. The evaluation of each $u_{i,j}^{(k+1)}$ corresponding to red nodes involves the values of black nodes only, and vice versa.



● black node where i+j is even
○ red node where i+j is odd
◍ boundary node

Fig. 1 Red-black ordering technique

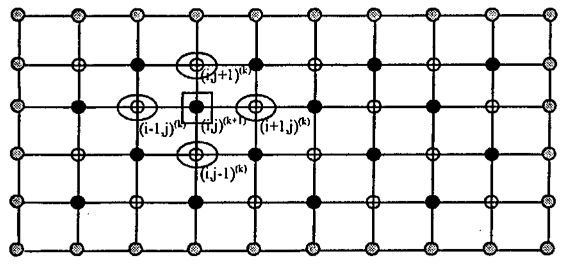
Based on the red-black ordering technique, the approximation $u_{i,j}^{(k+1)}$ can be updated in a different order suggested by Eq. (5). Each iteration of this method consists of two phases: (1) updating all the red values first and then (2) updating all the black values. The two phases are illustrated by Fig. 2(a) and Fig. 2(b), respectively.

For red nodes where $i + j$ is odd, we have

$$u_{i,j}^{(k+1)} = u_{i,j}^{(k)} + \frac{\omega}{4}(u_{i,j-1}^{(k)} + u_{i,j+1}^{(k)} + u_{i-1,j}^{(k)} + c_4 u_{i+1,j}^{(k)} - 4u_{i,j}^{(k)} - h^2 f_{i,j}) \quad (6)$$

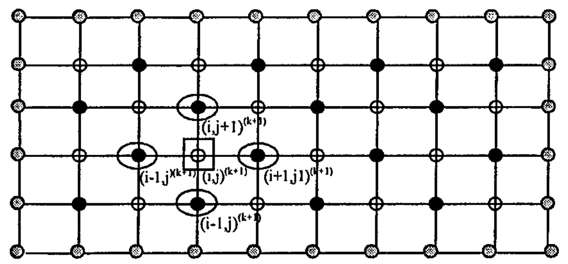
For black nodes where $i + j$ is even,

$$u_{i,j}^{(k+1)} = u_{i,j}^{(k)} + \frac{\omega}{4}(u_{i,j-1}^{(k+1)} + u_{i,j+1}^{(k+1)} + u_{i-1,j}^{(k+1)} + u_{i+1,j}^{(k+1)} - 4u_{i,j}^{(k)} - h^2 f_{i,j}) \quad (7)$$



▣ black node to be to updated
ⓞ red nodes used to update black nodes

(a) phase 1



⦿ black nodes used to update red nodes
▢ red node where i+j is odd

(b) phase 2

Fig. 2. Two phases of red-black implementation of the SOR algorithm. (a) Phase 1: updating the values of the black nodes. (b) Phase 2: updating the values of the black nodes.

In the phase 1, the computation of all red nodes depends on the values of all black nodes which are already available in the Phase 2 at the last iteration. Thus, the computation can be partitioned into a number of independent tasks and performed by multiprocessors. In the phase 2, the computation of all black nodes depends on the values of all red nodes which have been computed in the Phase 1. Similarly, it can also be partitioned into a number of independent tasks and performed by multiprocessors, as described in section 2.3. Multicolor SOR methods can be derived in a similar way.

2.3 Domain decomposition

Since the SOR method is applied for all nodes in the computation domain, an intuitive way to implement it in parallel is to divide the nodes into a number of subsets and each process performs the computation on one subset of nodes. This approach results in a domain decomposition. A rectangular computational domain can be partitioned into a number of subdomains using either row block partition or checkerboard partition. In the checkerboard partition, each process needs to communicate with several (up to four) adjacent processes and leads to more communication cost. In the row block partition, each process communicates with at most two processes and the communication pattern is simple as we will see in the next section. Thus, a row block domain decomposition method is employed in this study. Without loss of generality, the n rows of the mesh are divided evenly into p consecutive blocks, where p is the number of processes used. The entire computation is partitioned into p tasks, each of which is assigned onto one process for execution. There are a total of mn/p nodes in each subdomain. The two adjacent processes must exchange the data of their local boundary nodes.

2.4 Interprocess communication

To design an efficient parallel algorithm, we need to reduce both the computation cost of each task and the interprocess communication cost to achieve high performance. At the end of each phase of red-black SOR algorithm, all processes are synchronized. Based on the row black domain decomposition, row distribution is done to ensure that each processor gets an even amount of rows in order to balance the computational load. Each process only needs to communicate with its adjacent processes and sends them the most recent values on the internal boundary, as shown in Fig. 3. At the end of Phase 1, the process P_q sends the value of the red node (i,j) to the adjacent process P_{q+1} and the P_{q+1} sends the values at red nodes $(i-1,j-1)$ and $(i+1,j-1)$ to the nodes $(i-1,j)$ and $(i+1,j)$ on process P_q. At the end of Phase 2, the communication is quite similar except that the direction of communication is reversed.

Internal boundary rows are communicated using a non-blocking MPI_Isend and a blocking MPI_Recv. This ensures that deadlocks do not occur. Additionally, the root process P_0 containing the top rows only needs to communicate with the process P_1. The last process P_{p-1} containing the last groups of rows only needs to communicate with the process P_{p-2}. Additionally, convergence is calculated once per iteration using MPI_Reduce and its communication cost is $t_w \log p$. Ignoring the per-hop time and start-up time, the total communication cost in Phase 1 and 2 of red-black SOR parallel algorithm for each iteration is derived as given as follows

$$\begin{aligned} C &= C_{phase1} + C_{phase2} + C_{reduce} \\ &= t_w m(p-1) + t_w m(p-1) + t_w \log p \end{aligned} \tag{8}$$

where p is the total number of processes, m is the number of node on the inner boundary which is also equal to the node number in x direction, t_w is the transfer time per word, and C is total communication cost which is the sum of the send-receive communication costs in phases 1 and 2, and the communication cost of MPI_Reduce operation at the end of each iteration.

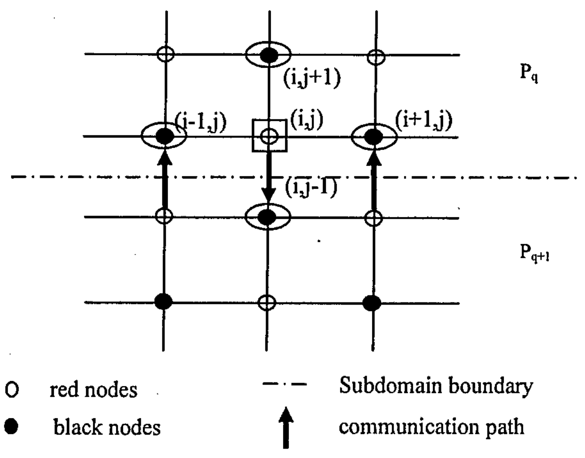


Fig. 3 Communications between two adjacent processes (Phase 1)

2.5 Parallel SOR algorithms

Based on the above design and analysis, the algorithm of parallel SOR with domain decomposition is developed, as shown in Algorithm 1.

```
procedure Parallel_SOR(My_ID, $\omega$, U_Curr, U_Next)
begin
  ...
  while (! isConvergent & iteration < max_iter) {
    /* update U_Curr which holds solution from
        previous iteration */
    U_Curr = U_Next;
    if (My_ID != ROOT)
      {above_row = get_above(U_Curr);}
    if (My_ID != LAST)
      {below_row = get_below(U_Curr);}
    /* Phase 1: solve for red values using black
    values from the previous iteration */
    solve_red(U_Curr, U_Next, above_row,
    below_row, $\omega$);

    /* Phase 2: solve for black values using red
    values for this iteration */
    solve_black(U_Next, U_Next, above_row,
    below_row, $\omega$);

    /* determine if it is convergent*/
    error_local = Compute_Error();
    if (My_ID = = ROOT) {
        isConvegent =
            convergence_control(err_local);
  } // end while
  gather partial result U_Next from all processes
end Parallel SOR algorithm
```

Algorithm 1 Parallel SOR algorithm with domain decomposition for distributed memory platform (Linux cluster), in which My_ID is the rank of each process, U_Curr represents $u^{(k)}$, U_Next represents $u^{(k+1)}$, and above_row and below_row are the two boundary rows in the subdomain used for communication

In this algorithm, the initial global data are scattered onto multiple processors and each processor only stores its local data. At the end of the phases 1, synchronization is enforced since the most recent values of all red nodes will be used for computation of black nodes in phase 2. Fig. 2 shows that the update of the values of all red nodes can be performed in parallel because the values of all black nodes are available and the tasks corresponding to different subdomains are independent, which simplifies the parallel implementation. The same procedure is applied for updating the values of all black nodes in phase 2. It is seen that for each phase of the SOR algorithm, the parallel computation is quite similar to the Jacobi method. Synchronization is also needed at the end of phase 2 for the same reason it is used with phase 1. Since the parallel SOR is an iterative algorithm, at the end of each iteration, the parallel computation is synchronized. The ROOT process determines whether the computation is convergent or not. If it is not convergent, the process continues and goes to the next iteration. Convergence is tested using the following criterion

$$\left\| u^{(k+1)} - u(k) \right\|_2 < \varepsilon \qquad (9)$$

where $u^{(k+1)}$ is the nodal solution for the latest iteration, $u^{(k)}$ is the nodal solution for the immediately past iteration, and ε is the convergence tolerance. The ROOT process sums up the squared difference of the current and new value at each node, and takes the square root of the total sum to test convergence.

3. Implementation

Many programming languages and libraries have been developed for explicit parallel programming. They differ in the view of the address space, degree of synchronization and multiplicity of programs. The message passing interface (MPI) is a standard for writing message passing programs. MPI was originally targeted for distributed memory systems, such as a cluster of PCs, but it is supported on virtually all high performance computing (HPC) platforms, including shared memory platforms, e.g., SGI Origin. In this paper, MPI is chosen for developing parallel computing applications because of its standardization, portability, performance, functionality and availability. The subroutines and functions in MPI are called from C code. The serial algorithm is implemented first, and then modified to support parallel computing. Both serial and parallel implementations produce the same convergent results.

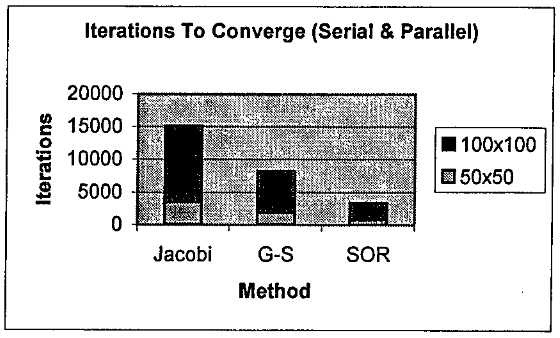
For most iterative scientific computing applications, when data decomposition technique is employed, all processes execute the same code but perform on different data sets. This is the single program and multiple data (SPMD) paradigm used for implementing parallel algorithms. Parallelism in MPI is explicit so developers need to consider how different processes perform their task and communicate with each other in the parallel algorithm design.

To validate the parallel SOR algorithm and evaluate its performance in Linux cluster, the connventional parallel Jacobi method is also implemented using C and MPI. The Jacobi method is described by the Eq. (3), in which the new value $u_{i,j}^{(k+1)}$ of any node at current iteration depends only on the old values $u_{i-1,j}^{(k)}$, $u_{i,j-1}^{(k)}$, $u_{i+1,j}^{(k)}$ and $u_{i,j+1}^{(k)}$ of the four neighboring nodes at the previous iteration. After a row block domain composition is applied for the entire computation domain, the task corresponding to each subdomain is to update the values of all inner nodes in this subdomain. Obviously, these tasks are independent and can be easily implemented in parallel. If the parameter ω in the parallel SOR algorithm is equal to 1, e.g., ω=1, the parallel SOR reduces to the Gauss_Seidel parallel algorithm. The parallel G-S is implemented in the same as the parallel SOR except for the value parameter ω.

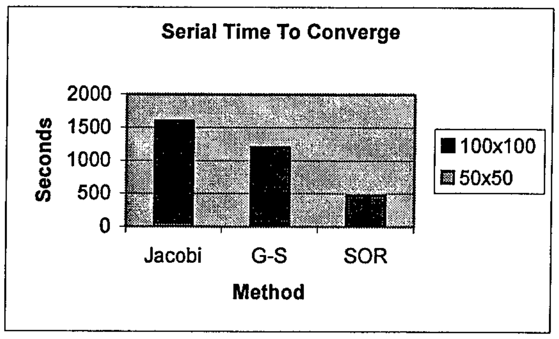
All performance tests were conducted on eight dual processor 2.6ghz 32 bit Intel Xeons, totaling 16 processors in all. The parallel computing platform containing these computers was a 100mbs switched Ethernet bus network local area network.

4. Performance Results

The SOR, G-S and Jacobi iterative methods are implemented in both serial and in parallel. Globalized output files were validated for both serial and parallel implementations of each method, and were found to be identical for all three methods on the domain with a 50×50 mesh and a 100×100 mesh. All three methods agree to the 1000th place. Fig. 4 shows the comparison of the number of iterations needed to converge and the total time each method takes to converge when run in serial. In SOR method, $\omega=1.9$ and in G-S method, $\omega=1.0$. Among the three iterative methods, SOR is clearly the fastest method in terms of serial time and the number of iterations.



(a)

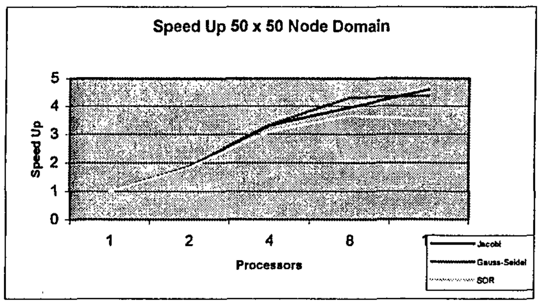


(b)

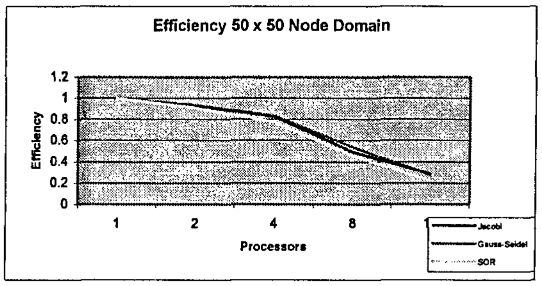
Fig. 4. Performance comparisons of the Jacobi, G-S ($\omega=1.0$), and SOR ($\omega=1.9$)iterative methods. (a) number of iterations with respect to different methods. (b) serial time with respect to different methods.

Performances are evaluated in terms of speedup and efficiency. The speedup S is defined as $S=T_s/T_p$ and the efficiency E is defined as $E=S/P$ where T_s and T_p are serial execution time and parallel execution time, respectively, and P is the number of processors. The performance results with respect to the 50x50 and 100x100 meshes and three methods are given in Fig. 5 and Fig. 6.

From Fig. 5(a) and 6(a), when the number of processors increases, the speedup increases. The actual speedup is smaller than the ideal speedup because the communication cost is relatively higher when implemented and executed on a Linux cluster, compared with the case when executed on a share memory platform. From Fig. 5(b) and 6(b), it is seen that when more processors are used for parallel computation, the communication cost increases, as given by Eq. (8), and the efficiency decreases.

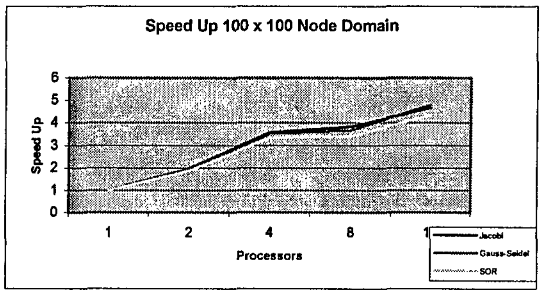


(a) Speedup

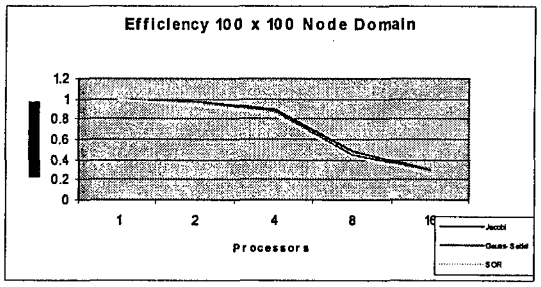


(b) Efficiency

Fig. 5. Speedup and efficiency with respect to three methods for the 50×50 mesh



(a) Speedup



(b) Efficiency

Fig. 6. Speedup and efficiency with respect to three methods for the 100×100 mesh

5. Conclusions

The paper designs the parallel algorithm for red-black SOR iterative method with domain decomposition and compares it with the parallel Jacobi method and G-S method. All three parallel iterative methods are implemented in C and MPI and executed on a Linux cluster with eight dual processor 2.6ghz 32 bit Intel Xeons, totaling 16 processors. The computers are connected by a 100mbs switched Ethernet bus network. The performance shows that, of the three iterative methods, SOR converges fast with a properly chosen parameter ω, e.g. $\omega = 1.9$. The speedup of the three methods increases but the efficiency decreases when the number of processors increases. In addition, the speedup and efficiency plots are quite similar for 50x50 and 100x100 meshes. The communication cost increases with an increase of the number of processors so the speedup of these methods are smaller when executed on a Linux cluster than that when executed on SGI share memory platform.

Acknowledgements. The research is partially supported by the Office of Naval Research, Naval Ocean Modeling Program, (Award No. N0003905WXFR146).

6. References

[1] T. L. Freeman and C. Phiilips, *Parallel Numerical Algorithms*, Prentice Hall International, 1992

[2] D. L. Harrar II, "Ordering, multicoloring, and consistently ordered matrics, *SIAM J. Matrtix Anal. Appl.* 14(1), 259-278 (1993).

[3] R. G. Melhem and K. V. S. Ramarao, "Multicolor ordering of sparse matrices resulting from irregular grid," *ACM Transaction on Math. Software*, 11, 117-138, (1988).

[4] I. R. Yavneh, "On red-black SOR smoothing in multigrid," *SIAM J. Sci. Comput.* 17(1), 180-192 (1995).

[5] D. Xie and L. Adams, "New parallel method by domain partitioning," *SIAM J. Sci. Comput.*

[6] W. Niethammer, "The SOR method on parallel computers," *Numer. Math.* 56, 247-254 (1989).

[7] D. J. Evans, "Parallel SOR iterative methods," *Parallel Computing*, 1, 3-18 (1984).

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 07-06-2006	**2. REPORT TYPE** Conference Proceedings (not refereed)	**3. DATES COVERED** *(From - To)*

4. TITLE AND SUBTITLE Parallel SOR Iterative Algorithms and Performance Evaluation on a Linux Cluster	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER 0603207N
6. AUTHOR(S) Chaoyang Zhang, Hong Lan, Yang Ye, Brett D. Estrade	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER 73-7625-05
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Research Laboratory Oceanography Division Stennis Space Center, MS 39529-5004	**8. PERFORMING ORGANIZATION REPORT NUMBER** NRL/PP/732005-5175
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Space & Naval Warfare Systems Command 2451 Crystal Dr. Arlington, VA 22245-5200	**10. SPONSOR/MONITOR'S ACRONYM(S)** SPAWAR
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release, distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The successive over-relaxation (SOR) iterative method is an important solver for linear systems. In this paper, a parallel algorithm for the red-black SOR method with domain decomposition is investigated. The parallel SOR algorithm is designed by combining the traditional red-black SOR and row block domain decomposition technique, which reduces the communication cost and simplifies the parallel implementation. Two other iterative methods, Jacobi and Gauss-Seidel (G-S), are also implemented in parallel for comparison. The three parallel iterative algorithms are implemented in C and MPI (Message Passing Interface) for solving the Dirichlet problem on a Linux cluster with eight dual processor 2.6ghz 32 bit Intel Xeons, totaling 16 processors. The performances of the three algorithms are evaluated in terms of speedup and efficiency.

15. SUBJECT TERMS

Parallel algorithm, successive over-relaxation (SOR) iteration, Linux cluster, message passing interface (MPI)

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Brett. D Estrade
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			
Unclassified	Unclassified	Unclassified	UL	7	**19b. TELEPHONE NUMBER** *(Include area code)* (228) 688-4151

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18